Assignment 2

[1]:
```
document1 = "The quick brown fox jumped over the lazy dog."
document2 = "The lazy dog slept in the sun."
```

[2]:
```
# Convert each document to lowercase and split it into words
tokens1 = document1.lower().split()
tokens2 = document2.lower().split()
print(f"Token1 : {tokens1}")
print(f"\n\nToken2 : {tokens2}")
```

```
Token1 : ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog.']


Token2 : ['the', 'lazy', 'dog', 'slept', 'in', 'the', 'sun.']
```

[3]:
```
# Combine the tokens into a list of unique terms
terms = list(set(tokens1 + tokens2))
print(f"Terms : {terms}")
```

```
Terms : ['fox', 'the', 'in', 'over', 'slept', 'dog', 'lazy', 'jumped', 'brown', 'sun.', 'quick', 'dog.']
```

[4]:
```
# Create an empty dictionary to store the inverted index
inverted_index = {}

# For each term, find the documents that contain it
for term in terms:
    documents = []
    if term in tokens1:
        documents.append("Document 1")
    if term in tokens2:
        documents.append("Document 2")
```

```
for term in terms:
    documents = []
    if term in tokens1:
        documents.append("Document 1")
    if term in tokens2:
        documents.append("Document 2")
    inverted_index[term] = documents

print(f"Inverted Index Dictionary : \n{inverted_index}")
```

```
Inverted Index Dictionary :
{'fox': ['Document 1'], 'the': ['Document 1', 'Document 2'], 'in': ['Document 2'], 'over': ['Document 1'], 'slept': ['Document 2'], 'dog':
['Document 2'], 'lazy': ['Document 1', 'Document 2'], 'jumped': ['Document 1'], 'brown': ['Document 1'], 'sun.': ['Document 2'], 'quick': ['D
ocument 1'], 'dog.': ['Document 1']}
```

[5]:
```
# Print inverted index
for term, documents in inverted_index.items():
    print(f"{term} -> {', '.join(documents)}")
```

```
fox -> Document 1
the -> Document 1, Document 2
in -> Document 2
over -> Document 1
slept -> Document 2
dog -> Document 2
lazy -> Document 1, Document 2
jumped -> Document 1
brown -> Document 1
sun. -> Document 2
quick -> Document 1
dog. -> Document 1
```

[1]:
```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from statsmodels.stats.multitest import multipletests
import matplotlib.pyplot as plt
import seaborn as sns

# Example RNA-Seq data (replace with actual dataset)
data = {
    'Gene': ['BRCA1', 'TP53', 'EGFR', 'MYC', 'PIK3CA', 'AKT1', 'PTEN', 'KRAS', 'NRAS', 'CDK2'],
    'Control_1': [520, 180, 350, 620, 400, 210, 500, 600, 320, 410],
    'Control_2': [510, 175, 340, 610, 410, 220, 510, 590, 310, 405],
    'Treatment_1': [700, 210, 500, 740, 480, 190, 550, 650, 300, 500],
    'Treatment_2': [690, 205, 510, 730, 490, 180, 560, 640, 290, 495]
}

# Convert the data to a DataFrame
df = pd.DataFrame(data)
df.set_index('Gene', inplace=True)

# Preview the data
print("Initial Data:")
print(df)
```

```
Initial Data:
        Control_1  Control_2  Treatment_1  Treatment_2
Gene
```

```
Initial Data:
       Control_1  Control_2  Treatment_1  Treatment_2
Gene
BRCA1        520        510          700          690
TP53         180        175          210          205
EGFR         350        340          500          510
MYC          620        610          740          730
PIK3CA       400        410          480          490
AKT1         210        220          190          180
PTEN         500        510          550          560
KRAS         600        590          650          640
NRAS         320        310          300          290
CDK2         410        405          500          495
```

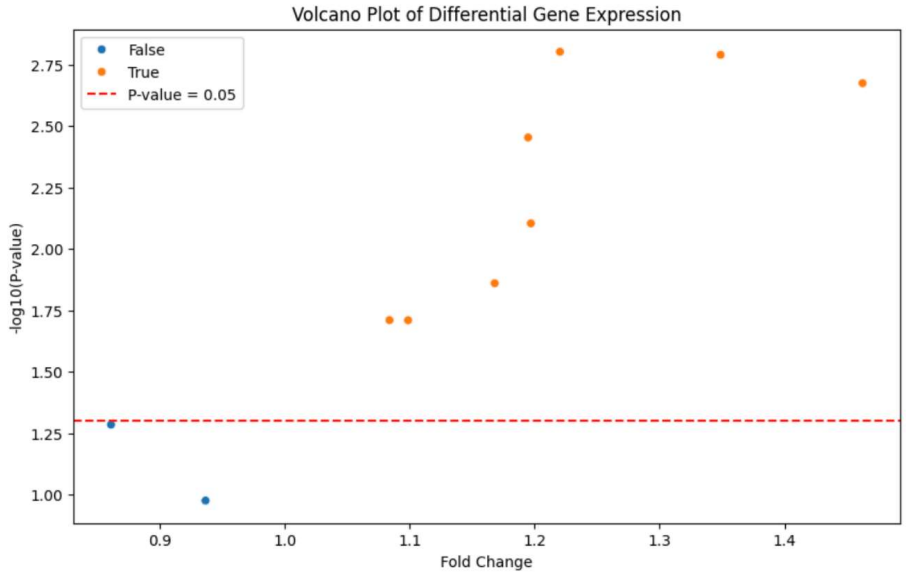
[2]:
```
# Add a column representing the condition (0 = Control, 1 = Treatment)
conditions = ['Control', 'Control', 'Treatment', 'Treatment']

# Perform log transformation to stabilize variance (common in RNA-Seq analysis)
df_log = np.log2(df + 1)

# Differential Expression Analysis using linear regression
results = []

for gene in df_log.index:
    # Response variable (expression levels)
    y = df_log.loc[gene].values

    # Independent variable (condition: Control vs Treatment)
    X = pd.get_dummies(conditions, drop_first=True)
```

```
    # Add intercept
    X = sm.add_constant(X)

    # Fit the model
    model = sm.OLS(y, X).fit()

    # Store the gene, fold change (coef), p-value
    fold_change = 2 ** model.params[1]  # Convert log2 fold change back to linear scale
    p_value = model.pvalues[1]
    results.append([gene, fold_change, p_value])

# Convert results to a DataFrame
results_df = pd.DataFrame(results, columns=['Gene', 'Fold_Change', 'P-value'])

# Adjust for multiple testing using the False Discovery Rate (FDR)
results_df['Adj_P-value'] = multipletests(results_df['P-value'], method='fdr_bh')[1]

# Display results
print("\nDifferential Expression Results:")
print(results_df)
```

```
print(results_df)
```

```
Differential Expression Results:
     Gene  Fold_Change   P-value  Adj_P-value
0   BRCA1     1.348866  0.001621     0.007047
1    TP53     1.168098  0.013791     0.022986
2    EGFR     1.462509  0.002114     0.007047
3     MYC     1.194817  0.003518     0.008794
4  PIK3CA     1.197072  0.007866     0.015731
5    AKT1     0.861031  0.051897     0.057664
6    PTEN     1.098823  0.019515     0.024394
7    KRAS     1.083898  0.019489     0.024394
8    NRAS     0.936692  0.105764     0.105764
9    CDK2     1.220326  0.001575     0.007047
```

[3]:
```
# Volcano plot for visualizing fold change vs significance
plt.figure(figsize=(10, 6))
sns.scatterplot(data=results_df, x=results_df['Fold_Change'], y=-np.log10(results_df['P-value']), hue=results_df['Adj_P-value'] < 0.05)

plt.axhline(y=-np.log10(0.05), color='red', linestyle='--', label='P-value = 0.05')
plt.xlabel('Fold Change')
plt.ylabel('-log10(P-value)')
plt.title('Volcano Plot of Differential Gene Expression')
plt.legend()
plt.show()
```



[ ]: Akanksha Pawar BEAD21235

Assignment 1

[1]:
```python
import re
```

[3]:
```python
# Function to find motifs in a DNA sequence
def find_motifs(sequence, motif):
    matches = re.finditer(motif, sequence)
    positions = [match.start() for match in matches]
    return positions
```

[5]:
```python
# Function to calculate GC content in a DNA sequence
def calculate_gc_content(sequence):
    gc_count = sequence.count('G') + sequence.count('C')
    total_bases = len(sequence)
    gc_content = (gc_count / total_bases) * 100
    return gc_content
```

[7]:
```python
# Function to identify coding regions (example: start codon 'ATG' and stop codon 'TAA')
def identify_coding_regions(sequence):
    start_codon = 'ATG'
    stop_codon = 'TAA'
    coding_regions = []
    start_positions = find_motifs(sequence, start_codon)
    stop_positions = find_motifs(sequence, stop_codon)

    for start in start_positions:
        for stop in stop_positions:
            if stop > start and (stop - start) % 3 == 0:
                coding_regions.append((start, stop + 2))

    return coding_regions
```

[9]:
```python
# Example DNA sequence (replace with your own sequence)
dna_sequence = "ATGGCCTAAATGGGCTAA"
```

[11]:
```python
# Find motifs
motif_to_find = "ATG"
motifs_found = find_motifs(dna_sequence, motif_to_find)
print(f"Motifs found: {motifs_found}")
```

```
Motifs found: [0, 9]
```

[13]:
```python
# Calculate GC content
gc_content = calculate_gc_content(dna_sequence)
print(f"GC content: {gc_content}%")
```

```
GC content: 44.44444444444444%
```

[15]:
```python
# Identify coding regions
coding_regions = identify_coding_regions(dna_sequence)
print(f"Coding regions: {coding_regions}")
```

```
Coding regions: [(0, 8), (0, 17), (9, 17)]
```

[ ]: